Filesystem Integrity Check Design

Author	Date	Description of Document Change	Client Approval By	Client Approval Date
Kalpak Shah	03/13/09	Initial draft (with edits by John Dawson)		
Kalpak Shah	03/15/09	Format and editing changes by John Dawson		
Kalpak Shah	03/15/09	Incorporate comments from Kalpak - JKD		
Andreas Dilger	06/01/09	Update based on comments		
Andreas Dilger	06/15/09	Editing & copyright		

1. Introduction

The HPCS filesystem verification goal is to check the integrity of a Lustre filesystem with 1 trillion files (10^{12} files) in 100 hours. The integrity check should ensure the internal consistency of both the back-end filesystem (ldiskfs, ZFS) and the Lustre filesystem inter-node consistency. A filesystem with this many files will itself take in the neighborhood of 4PB of raw storage just to store the metadata, once overhead such as metadata replication and internal filesystem overhead is taken into account.

It should be possible to do the filesystem integrity checking on a live filesystem (online), due to the uptime goal of 99.99% leaving less than 1h of downtime per year. It would be acceptable if we need to take individual servers off-line for back-end filesystem check should they be severely corrupted, but the rest of the filesystem should not be affected by this operation.

The ZFS back-end filesystem can verify the data and metadata integrity while the filesystem is mounted and in use. In normal operation the filesystem idle cycles can be used to opportunistically perform continuous or periodic integrity checking of the local ZFS filesystem. Similarly, the Lustre distributed integrity checking can validate the consistency of any distributed object without doing a filesystem scan and can be run during filesystem idle time. This will provide a high degree of confidence in the on disk data integrity even in the face of ongoing silent data corruption at the device level.

In the case of a catastrophic filesystem failure, a full filesystem integrity check might be needed. Due the extremely high file verification rate, approaching 3 million files per second, this requires parallel checking of the filesystem using a considerable number MDTs in a CMD configuration, depending on the actual RPC rate that each server can handle.

Fortunately, because Lustre is an object-based filesystem and each server maintains a self-consistent internal filesystem, at the Lustre level there are only a limited number of corruption scenarios that can be hit. The isolation of the backing on-disk filesystem from the clients ensures that misbehaving clients cannot introduce corruption into the backing filesystems.

2. Requirements

In order to achieve the goals for this feature, additional Lustre features will have to be completed and an appropriate hardware configuration that meets the raw throughput and IOPS requirements will have to be used.

2.1. Many of the limits needed by this project will only be available when we can use the ZFS Data Management Unit (DMU) as the back-end storage for Lustre MDT and OST filesystems.

2.2. Even with the DMU back end, we will need multiple metadata servers to be able to have 1 trillion files in the filesystem. This implies that the Clustered Metadata (CMD) working with Lustre-DMU is also a requirement.

2.3. Storage that is capable of enough IOPS/throughput needed by the integrity checks. This includes fast enough secondary storage as well as enough RAM to fit the consistency checking data structures in memory.

2.4. In order to reduce or avoid repeat traversal of the MDT backing filesystem, additional features need to be added into ZFS/DMU to integrate the local filesystem consistency checking with the distributed checking. ZFS scrubbing involves validating the checksums of each block while traversing the filesystem tree from top to bottom. This traversal is done block by block, and the type of block (directory block, extended attribute, inode block, etc.) can be determined from its block pointer and used to do the consistency checking of each block accordingly. By introducing *scrub callbacks*, and scrubbing only metadata blocks, Lustre can validate the distributed coherency for each block immediately after it has been validated by the underlying filesystem. For ldiskfs-based filesystems the scrub callbacks can also be implemented via a simple inode table and directory block traversal.

2.5. The design should not use any on-disk databases since such a design would not have sufficient performance. We should use the filesystem itself as the database. Use of in-memory data structures is acceptable as long as the upper bounds of memory usage can be defined.

2.6. Hardware requirements to meet these HPCS goals:

Checking 1 trillion files in 100 hours means that we need to continuously check both the local and distributed coherency of nearly 3 million files per second in aggregate across all MDTs and OSTs. While local e2fsck checking has been observed at rate of 140k files/sec, there are the following advantages with e2fsck compared to the distributed lfsck checking:

- While e2fsck is running the filesystem is unmounted,

- No network communication is needed for e2fsck.

In ZFS, inodes are more difficult to read in disk-order, the filesystem is going to be up during the check so we will not have control over all possible IOPS/throughput and we will need network communication. Also ZFS requires many more lookups than ldiskfs to get information since it is a tree-based filesystem. Considering this, we will likely be able to process fewer files/second for ZFS compared to ldiskfs.

Assuming that we can do the distributed checking of 25,000 files/sec for each MDT we will need approximately 128 clustered metadata servers to finish the check in 100 hours. Each MDT would manage about 32TB of raw space in a RAID-1 configuration. Using 40 800GB SSD disks for the high IOPS rate would ensure that the underlying disk is not the bottleneck. However, the algorithms will still be designed in order to minimize seek activity for non-SSD storage.

In order to reduce the disk reads and seeks for ZFS/DMU, there are several approaches that can be combined to improve performance:

- Use a large Adaptive Replacement Cache (ARC). This way, the top-level indirect blocks of the directory tree and inode tree will be cached in most cases. The MDS should have RAM in the range of 32GB-64GB considering each MDS will serve around 8 billion (1 trillion/128 MDS) files.

- Do the checking in an order that is determined by the on-disk locality (e.g. sequential on-disk order), rather than a predetermined order (e.g. namespace tree order). This can be achieved by allowing the ZFS scrub to select an optimal traversal of the filesystem metadata, and then leveraging that for the distributed Lustre coherency checking via the *scrub callbacks*.

- Avoid re-reading data from disk, by reusing the same buffers that the ZFS scrub has read from disk, and doing all or a majority of Lustre checks directly on the supplied buffer instead of doing a multi-pass algorithm (e.g don't do the entire local ZFS integrity checking first, then run Lustre integrity checking separately).

3. Functional Specification

This project can be broadly divided into the following sub-projects:

3.1. Online Lustre consistency checking

The current Lustre filesystem integrity checking tool, ***lfsck**,* requires a preliminary local ldiskfs check via e2fsck, which creates a set of external databases of all the information on the MDT and the OSTs. Furthermore while lfsck is running, it requires access to all the databases from a shared location. The time required for creation of the databases and running lfsck on them is prohibitive for very large filesystems due to large database sizes.

Therefore we plan to write a new tool online lfsck which will perform consistency checking of the Lustre filesystem while the filesystem is online. This is similar to scrubbing in ZFS - the only difference being that this scrubbing will validate filesystem structures and object connectivity.

3.1.1. OST Internal Consistency

Each OST and MDT backing filesystem has an internally consistent filesystem that can be verified using local checks. Items such as block allocation, local inode allocation, and some aspects of the directory structure can be checked internally and in parallel on each OST and MDT.

The OST backing filesystem has no externally visible namespace, only exporting objects via object IDs. Internally it has a very simple directory structure in order to map object IDs to local inode numbers, along with a small number of files that contain configuration, recovery information, and other internal state.

3.1.2. MDT-OST Consistency

Each MDT inode contains a layout attribute (LOV EA) that indicates which OST(s) and the object(s) thereon hold the file data. Each OST object contains an attribute called filter_fid that holds its object ID, and the MDT inode of which it is a member and its relative position within the file. The former allows an OST object to be recoverable even in case of catastrophic directory corruption.

The following conditions must be checked for each MDT inode to ensure MDT-OST inode-object consistency:

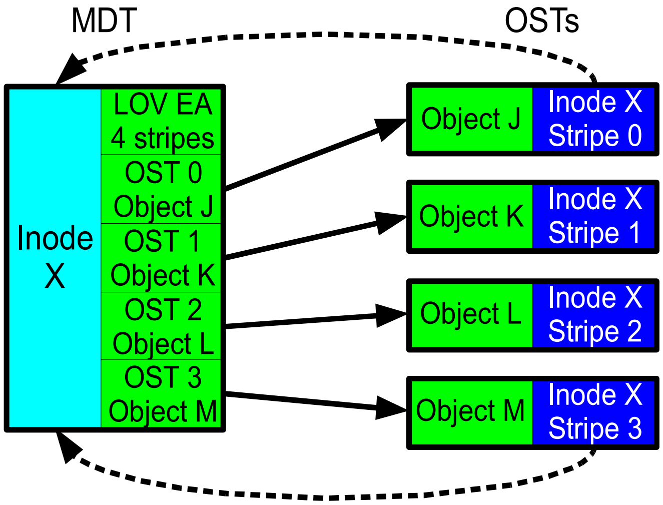


Figure 1: MDT-OST inode-object cross-references

3.1.2.1. All objects for each inode on the MDT should exist

3.1.2.2. Multiple MDS entries do not reference the same object

3.1.2.3. File size on each MDT inode matches the OST objects' size

3.1.2.4. Each object on the OST should belong to some file on the MDS

The first three checks can be done in a single pass during MDT inode traversal using MDT-to-OST RPCs, which will ensure that there are no user-visible problems in the filesystem. The MDT layout that determines which OST object(s) contain the data for that file. The OST object's back-pointer to the MDT inode allows each object to verify immediately that the correct MDT inode is being checked at the time of the MDT-OST verification is being done and avoids requiring any saved state to detect duplicate references.

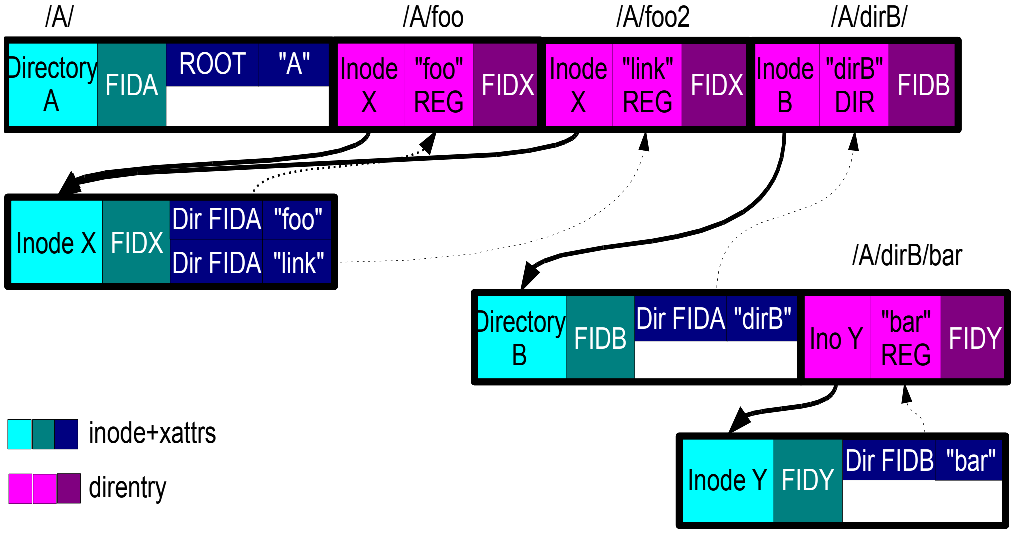
OST objects which are not referenced by the MDT do not introduce user-visible errors in the filesystem, but only cause space to be leaked. To find such unreferenced OST objects requires that all checked OST objects be tracked and unreferenced objects handled at the end of the MDT traversal, using about 1 bit of memory per object on the OST. For 1 trillion files with one object per file on 1024 OSTs, this needs 1 billion bits, or about 128MB of memory per OST.

3.1.3. MDT Internal Consistency

As with the OST filesystems, the MDT backing filesystem is an internally consistent filesystem, and can be verified locally for its internal metadata structure. The MDT backing filesystems contain almost exclusively directories and empty inodes with layout attributes for the regular files. The directory entries each contain the filename, the file type, the inode number, and the File Identifier (FID) number. Each inode also contains an attribute which identifies its FID. A separate FID-to-path attribute on each inode contains an array of parent directory FIDs along with the inode's filename in that parent directory.

Figure 2 shows an inode *X* with 2 hard links, /A/foo, /A/link, and a second sub-directory /A/dirB/ with a single /A/dirB/bar entry in it. Each of the directory entries provides the forward lookup from {parent, filename} to a specific inode. Unique to Lustre is the {parent, filename} duo in the path-to-FID attribute for either replication or integrity checking.

The FID number uniquely identifies each MDT inode even though there will be multiple MDTs making up a single filesystem namespace. The FIDs are mapped to specific MDTs using the *FID Location Database* (FLDB). Once the MDT is located, the *Object Index* (OI) is used to map a FID to a specific MDT-local inode. The on-disk structure of the FLDB and OI are internal implementation details. The FLDB will be redundant over a quorum of MDT nodes, and can be reconstructed during filesystem integrity checking. The OI is internally redundant, and can similarly be reconstructed during filesystem integrity checking.



When verifying a directory block's entries it is possible to directly verify that the inode has a back-pointer to that parent directory. By storing the parent FID and the corresponding filename in an array on each inode it is possible to immediately determine all of the hard links on each inode for a non-directory file, and in turn the file link count. For directories it is possible to verify which entries are subdirectories by using the type stored in each entry, and in turn this can be used to verify the link count on the directory inode itself.

3.1.4. MDT-MDT Consistency for CMD

In order to have the single CMD namespace traverse multiple filesystems, some directory entries are *remote entries* that reference an inode on another MDT. The inodes that are referenced by a remote directory entry are kept in a *proxy directory* on the remote MDT in order to keep the inode's link count consistent within that MDT filesystem. In each MDT filesystem there will be one or more proxy directories on each of the other MDTs. The proxy directories are not visible in the Lustre namespace, but allow a simple mechanism to track the remote inodes between each pair of MDTs.

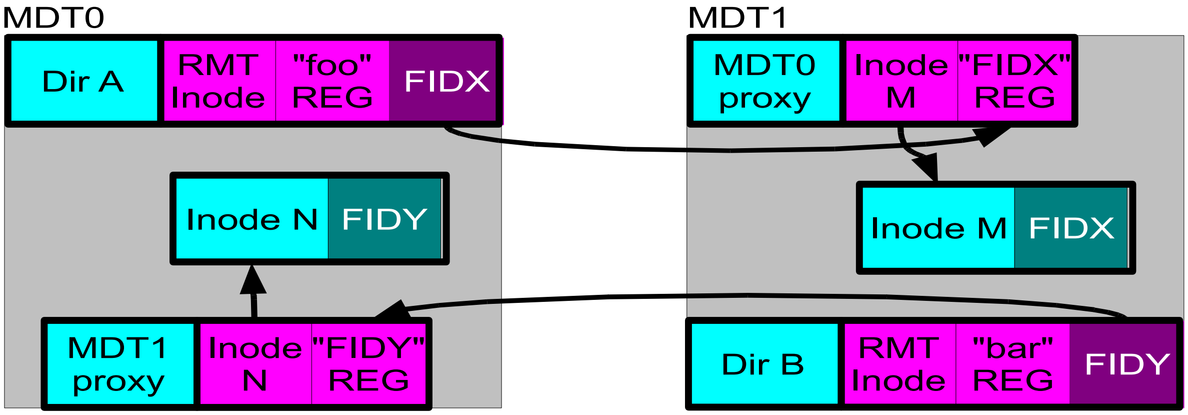


Figure 3: Inter-MDT cross-references

The following distributed Lustre checks must be done to ensure inter-MDT consistency.

3.1.4.1. Each directory entry references a valid local inode, or references a remote FID. The referenced inode has a corresponding FID-to-path reverse mapping entry in that inode that matches the directory entry.

3.1.4.2. The directory entry file type should match the inode file type.

3.1.4.3. For every FID-to-path entry on an inode there is a directory entry that references the inode. An empty FID-to-path table indicates an unreferenced inode.

3.1.4.4. The FID stored on each inode should map in the FID Location Database to the MDT on which the inode is located.

3.1.4.5. Regular files should have the inode link count equal to the number of items in the FID-to-path array.

3.1.4.6. Directories should only have a single FID-to-path entry that matches the ".." entry. The directory link count should equal the number of subdirectories, as determined by the validated directory entry file type, plus the parent reference. The user-visible directory link count of a split directory is computed from the sum of the slave directory link counts, and this allows the link counts of the individual slave directories to be validated internally and independently.

3.1.4.7. There should be no directory loops in the metadata namespace.

3.2. Design Premise

Initially we can design and implement an online integrity check for an ldiskfs back-end. The design and the Lustre inter-node coherency checking will also be usable with a ZFS back-end.

This requires we design a communication protocol between MDS-OSS and MDS-MDS which does consistency checking in an efficient manner.

3.2.1. For cases 3.1.2.1 - “All objects for each inode on each MDT should exist” and 3.1.2.2 - “Multiple MDS entries do not reference the same object”

We will have an inode iterator on the MDT which will scan through each used inode and the scrub callback for each inode will query the OSTs that hold data objects for the file and make sure that the object exists and that the filter_fid attribute of the OST object matches the MDT inode FID. If the object doesn’t exist then we recreate the object. Any duplicate or otherwise incorrect object reference by an MDT inode will result in the filter_fid attribute on the OST object and the MDT inode’s FID not matching. This problem can be detected and corrected during the MDT inode iteration.

We will aggregate queries for multiple inodes in a single RPC. This will save network bandwidth and the OST can also sort the objects and read in disk optimal order.

To make sure that checking can be restarted across reboots, we store the last checked inode persistently on each MDT in a file in the Lustre-internal part of the backing filesystem namespace.

3.2.2. For case 3.1.2.3 - “File size on each MDT inode matches the OST objects' size”

With the Size On MDS (SOM) feature the MDT caches an up-to-date total file size on the MDT inode. The update of the file size on the MDT inode is designed to be transactional, and should not become inconsistent with any combination of client, MDS, and OSS node reboots or recovery. Since the actual file size is determined by the OST objects there exists a possibility that the file size becomes inconsistent between the MDT copy and one or more of the OST objects. During MDT traversal the MDS should verify the SOM cached value against the objects in the file's layout. At least one of the OST objects should have a size that is exactly equal to the SOM cache value on the MDT. It is possible that some OST objects will have a smaller size due to sparse writes not going to all OSTs.

3.2.3. For case 3.1.2.4 - “Each object on the OST should belong to some file on the MDS”

This orphan detection cannot be done from the MDT inode iteration alone. Each OST needs to iterate over all of its objects and either:

- Do MDS_GETATTR RPCs on the MDT inode referenced filter_fid to verify that the specified MDT inode exists, and that this object is referenced at the stripe offset stored in the filter_fid. This has significant overhead as the same inode may be checked multiple times (once for each object on each OST), and the MDT inode accesses will be very random due to the OST-preferential ordering.

- Keep a bitmap (RB tree) of all in-use objects, wait for the MDS to issue OST_GETATTR RPCs for each object, validate the local object state, and clear the bit for the checked object. Any OST objects still in the in-use bitmap (i.e. never checked by the MDT) are orphans, though this should be verified by the OSS with an MDS_GETATTR RPC as above.

- Some combination of the two above mechanisms.

For ongoing background integrity verification (i.e. not during a full filesystem integrity check after some catastrophic event) it is likely that a low intensity OSS-driven scan is desirable to allow orphan object detection, because the MDS-driven scan will take a long time at a low background rate.

Destroying an OST object during a scan would also clear the bit for that object in the bitmap. This might happen either due to user actions on the filesystem being in use and available during a scan, or due to the MDT deleting an inode during its internal consistency checks after detecting an unreferenced inode.

For ldiskfs the filesystem iterator could be implemented by simply reading the inode bitmaps into an array at the start of the check (in ldiskfs inode order) then clearing the corresponding bit when the object is verified or unlinked (after object ID to local inode number conversion). For an unreferenced OST object (bit still set in the OST in-memory bitmap) its MDT inode can be verified as deleted on the MDT by using the filter_fid to send an MDS_GETATTR RPC to the MDS for that FID.

In the case of the ZFS back-end, the dnodes are not saved in disk order and hence it can potentially cause a great deal of disk seek activity if we iterate through the dnodes in serial order. We can solve this problem by having the ZFS scrub traverse only the internal metadata tree and log the inode block numbers, which can then be sorted and read in disk optimal order.

3.2.4. For cases 3.1.4.1 - “Each directory entry references a valid inode”, 3.1.4.2 - “Each directory entry has a valid file type”, and 3.1.4.4 - “The FID maps to the MDT on which the inode resides”

While checking the directory blocks in the MDT filesystem these properties can be validated. Each directory entry contains the filename, the file type, the local inode number (if any), and the Lustre FID.

- For directory root blocks, the ".." entry should match the FID-to-path attribute on the directory inode. If the directory is on another MDT from its parent the ".." entry should reference a proxy directory for the remote MDT.

- If the directory entry FID references the local MDT then there should be a local inode number in the directory entry. This can be verified with the local Object Index.

- If the directory entry FID references a remote MDT then the directory entry should be flagged as a remote entry.

• Each directory entry should have a corresponding entry in the FID-to-path array on the inode.

3.2.5. Verify that the link count for each directory

Checking link counts across MDTs could be very bad for performance if not implemented correctly. The design for CMD is to do the following:

- Directory entries that are referenced from a remote MDT should reside in a local "proxy parent directory" for that particular remote MDT.
- e2fsck/zfsck will understand when a directory entry refers to a remote object with a special flag.
- This gives us a central place to quickly verify the cross-MDT reference counts. The link count on MDT0's proxy inode for MDT1 should be the same as the number of entries in MDT1's proxy parent directory for MDT0 and vice-a-vera, repeated for all MDT pairs.

3.3. Online backend filesystem scrubbing

ZFS has a scrub command which can be used to check all the data in the pool for checksum consistency. This can be run while the filesystem is online. There are the following problems with scrubbing in ZFS and the solutions we will need:

3.3.1. ZFS scrub creates lots of IO and any filesystem activity becomes very slow during the scrub. The reason for this is that scrub also checks all the data blocks for consistency. We can modify ZFS scrub and add an option which will only scrub the metadata blocks, thereby reducing the IO done during scrubbing. This will also ensure that scrub takes much less time than it does now.

3.3.2. During the scrub we will traverse all the metadata blocks and the entire ZFS filesystem tree. It would be suboptimal to have to traverse the tree again for inode traversal require by the integrity check. To solve this problem we can add callbacks in ZFS scrub, so that the inode traversal is done during the scrub itself.

3.3.3. Scrub only checks for checksum consistency and does not check for namespace consistency or data structures like a full integrity check should. We can modify scrub to perform these functions.

ZFS currently depends on checksums for all its consistency checking. It does not verify space maps, directory tree or any other data structure integrity. This is acceptable in most cases since checksum mismatch indicates a corrupted block which is then corrected from the copies of the block normally stored on other disks. We will take up the issue with the ZFS team regarding creation of a ZFS check tool which can do filesystem checking at the ZFS level.

3.4. New ZFS Features

3.4.1. Callbacks in ZFS scrub

As mentioned earlier we can have callbacks for certain types of blocks - for example we can have a callbacks for each inode block, directory block, etc. to perform different checks for log information that can be used later. While doing the DMU traversal we need to save information about the blocks such as type of block, logical offset of block in file, etc. We may need to store some information within the block pointer itself for this.

3.4.2. Scrub only metadata blocks

The online checking should make sure that it does not slow down the normal filesystem operations to any appreciable extent. We can use parameters like adaptive timeouts, IO load on the servers to speed-up or throttle the filesystem checking speed.

4. Use Cases

4.1. Complete check on 1 trillion files in 100 hours

4.2. Online check should slow itself down when server is busy with filesystem IO and speed up when server has low load. This does not mean that we restart the process, we just slow it down and we do not lose any state.

4.3. Even after reboot, the integrity check should resume from where it left off.

4.4. All inconsistencies should be repaired in one scan of the filesystem.

4.5. If there is a missing OST at the time of the check, we should initially just skip the checks for the objects on that OST. There is no point to keep a list of the missing OST objects on the MDT since we will likely run the check again in the future, or if the check is still running when the missing OST comes back online and can begin verifying the remaining objects on the OST. The OST would still initialize an in-use object bitmap, cancel the ones that the MDS verifies during the rest of the MDS-directed integrity checking, and then do MDS_GETATTR RPCs to verify any unreferenced objects.

If the MDS-directed check is finished, then the OSS would just populate a bitmap of all in-use objects and check the MDT inodes for them, with the expectation that most of them are going to be valid.

4.6. This is a detailed description of possible errors from the integrity check and our corrective actions:

4.6.1. Multiple MDS entries should not reference the same object
Any duplicate object reference by an MDT inode will be caught because this will result in the filter_fid of the OST inode not matching. We will check if the filter_fid of the object matches a valid MDT inode and if so we understand that the earlier MDT inode is bad and we create a new object for it.

4.6.2. All objects for a file on the MDS should exist. If we find that an object is missing for some MDT inode, we recreate that object and add a reference to the MDT inode in the new object's filter_fid.

4.6.3. Each object on the OST should belong to some file on the MDS. If we complete the MDT inode traversal and have objects existing from the beginning of the check that were not referenced during the MDT traversal, or have completed an OSS-directed traversal, this indicates that we have found orphan objects. We can check the filter_fid for each object and do MDT_GETATTR for the MDT inode. If the MDT inode references this object then there is nothing to be done (likely a race between file creation and on the MDT and OST). Otherwise, if the back reference from the OST object does not match any MDT inode, or the inode does not reference this object, then we move the objects to the lost+found directory, or simply unlink it (as is set in the policy by the administrator).

5. State Management

5.1. State Invariants

5.2. Scalability & Performance

In order to make the solution scalable we have done away with on-disk databases and are moving towards maintaining in-memory structures and per-inode data. This will ensure that the checking scales with filesystem size. There is one exception to this where we need to maintain a bitmap on each of the OSTs to track which objects that have been checked. For example for 1 billion objects, we will need 128MB of RAM.

Online integrity checking will affect the performance of the filesystem slightly, but we can make sure that its communication and disk IOPS do not affect the filesystem performance more than what is acceptable to the administrator by throttling the the integrity checking communication and IOPS during heavy filesystem load. There will of course be a tradeoff between fast verification and available performance if the filesystem is always busy.

The rest of the performance details are mentioned in the Requirements section.

5.3. Recovery Changes

This task does not affect recovery in any way.

5.4. Locking Changes

During current lfsck, the state of the filesystem can change while the check is running and we may encounter some false positives. Now when we run our new online consistency checking we do not want to see lots of false positives, so we will need to some local locking, mostly read locks while checking inodes/dnodes. If we cannot get the read lock for an inode then we can check it later. In case of false positives due to our decision to avoid global locks, we can send a re-check query.

5.5. Disk Format Changes

We will need to store the last inode checked so that checking can resume from that point if machine reboots or crashes. We can also save some more status in a special status file.

5.6. Wire Format Changes

We will add a "GETATTR_PLUS" operation to the OST that will take as input an array of {objid,grp,mdsfid} and return an array of attributes for them. Clients or MDSes may want to use this for other reasons as well.

Conversely, the OST will want to also be able to do GETATTR_PLUS to have the MDS verify many inodes match OST objects at one time. This will be very similar to what readdir+ would do on the client, and it would be good to have common code/RPC for doing this. Even in the readdir+ case it may be useful for clients to first read the dirents, and then do the GETATTR_PLUS separately so that they only fetch the attributes for the inodes they do not already have.

5.7. Protocol Changes

5.8. API Changes

Existing APIs will not change.

5.9. RPC Order Changes

RPC order will not change.

6. Alternatives

- ZFS team to implement a ZFS filesystem checker (***zfsck***). If this is done, then we won’t need to modify ZFS scrub at all. Instead we can hook some of the checking code into zfsck to fetch us information. zfsck will be dedicated to the job of internal filesystem consistency checking.

- There was discussion about having a gigantic distributed MPI filesystem verification job that ran on the whole cluster and just loaded all the metadata into RAM and began crunching it, but the drawback is that (a) it makes the cluster unusable during this time, (b) it is complex to write, (c) it can't be run incrementally in the background during periods of low IO traffic.